Fall 2019

Queue: A Mobile Application for Collaborative Music Playlists

Vlad Mirea
vm63@zips.uakron.edu

Queue: A Mobile Application for Collaborative Music Playlists

Vlad Mirea

Department of Computer Science

Honors Research Project

Submitted to

The Williams Honors College
The University of Akron

Approved:

Date: 12-4-2019
Honors Project Sponsor (signed)

Yingcai Xiao
Honors Project Sponsor (printed)
Yingcai Xiao

Date: 12-4-19
Reader (signed)

Will S. Crissey, Jr.
Reader (printed)

Date: 12-5-19
Reader (signed)

Zhong-Hui Duan
Reader (printed)

Accepted:

Date: 12/6/2019
Honors Department Advisor (signed)

TIM O'NEIL
Honors Department Advisor (printed)

Date: 12/15/2019
Department Chair (signed)

TIM O'NEIL
Department Chair (printed)

Chapter 1

Introduction

Mobile application development is a complex and prosperous field that is evolving every day. For many tasks, it is more convenient for the consumer to use their smartphone over another device. For that reason, as an application developer, mobile app development is an important skill to possess.

In order to gain a better understanding of the mobile application development process while getting to apply software development concepts studied in my undergraduate courses, I developed Queue. Queue is an Android application that allows a host to create a lobby and allow theirs guests to join and add songs for the host to play.

1.1 Goals and Objectives

The purpose of this project was to leverage my studies and experiences in computer science to create a non-trivial, useful application while still researching and learning new tools and technologies. Throughout my career in software development, I have developed a number of software projects in many different languages but I have never created an application with actual commercial value. I wanted to challenge myself and create something that fills a need in the market and that I would actually use myself.

Queue was an idea a friend and I had discussed in the past but never ended up implementing. I knew that this would be a more complex application than most others I had worked on before, so it definitely fit the “non-trivial” criteria. Another reason that

swayed me towards this project idea was that I could work with something I love, music, and at the end I would have a finished project that my friends and I could use.

This project idea spawned out of a real need so my biggest priority when it came to features was satisfying the core requirements for functionality that my friend and I originally came up with. In its simplest form, that was a mobile app that allowed one person to host a lobby where others could add songs to be played by the host. While that may sound relatively simple at first, that still requires managing many different components, from vastly different topics of computer science, and making them work together seamlessly.

Chapter 2

Page Flow

2.1 Design

My first step was to create some designs of how I wanted the finished product to flow. The idea was to keep the flow as simple and intuitive as possible. It should be fast and easy to create or join a lobby; the user should not need to search multiple menus for their desired action or need to worry about how to connect to other devices. The final Queue flow is below.

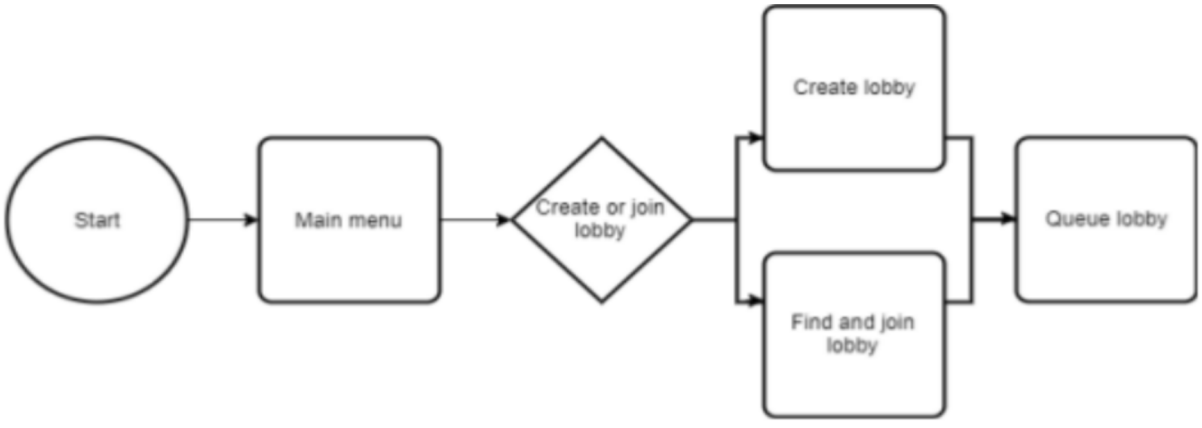

Figure 1: *A diagram depicting flow of creating or joining a Queue lobby*

2.2 Implementation

Each screen in the Queue app is implemented as an Android activity. Every activity has a lifecycle it goes through starting with activity creation when the activity is first launched, and ending with the activity being destroyed after terminating. An activity can go through this lifecycle many times in many different ways so proper handling of the activities' lifecycles is imperative.

When an activity is started for the first time, the onCreate() function is called. Here, all the preliminary, one-time setup is implemented. Before an activity is completely removed, it goes through its onDestroy() function. Here I clean up the things I set up in onCreate(). When the activity is no longer visible, the activity is stopped and onStop() is triggered. In this state, the Android operating system may kill the activity to free up memory for other activities. For that reason, I saved the activity state and paused non-critical

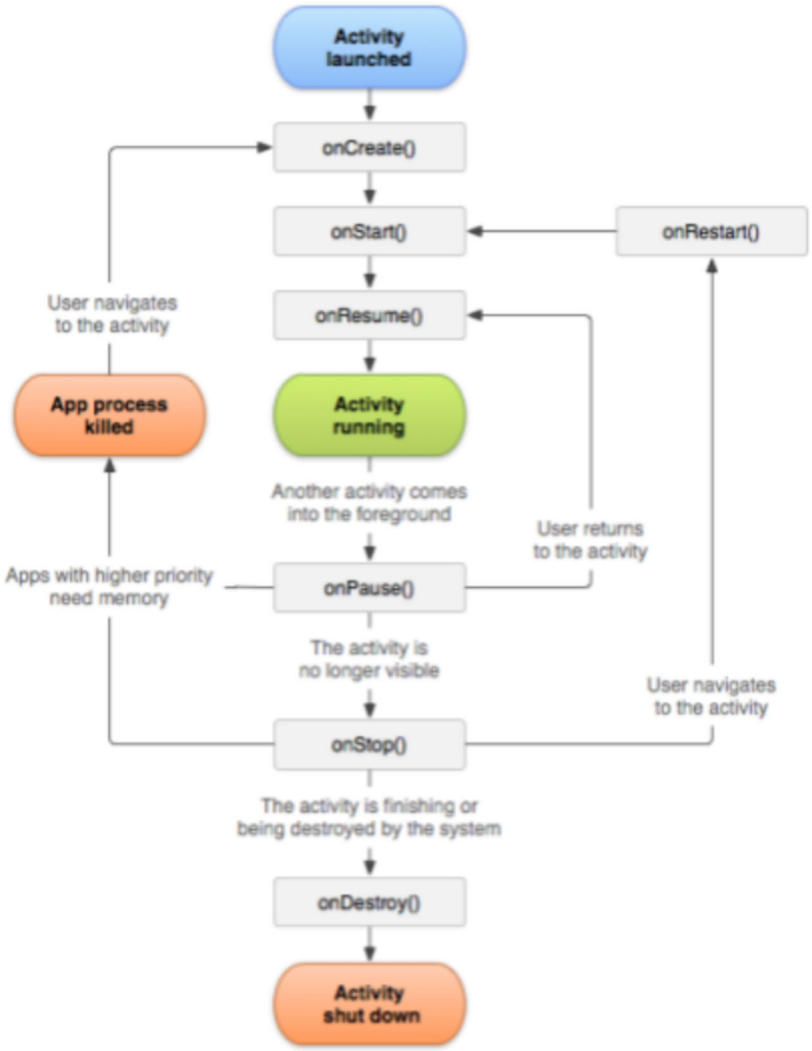

***Figure 2**: A simplified diagram of the activity lifecycle*

functions in this method. When the activity regains focus, it goes through onStart() where it resumes all normal functions.

In addition to handling interaction within a single activity, creating an app also requires starting and communicating with other activities. The main method I used to handle this is with intents. An intent is an abstract description of an operation to be performed. Within the intent, I specify the new activity to be started, as well as any data that I wish to pass. The Android operating system starts this new activity and passes along the intent. When the new activity starts, it can unpack the intent to receive the data from the previous activity.

Chapter 3

Queue Lobby

3.1 Design

The lobby screen is where Queue users will spend most of their time in. When in a lobby, users have numerous functions they may wish to perform or information they want to view. To provide the best user experience, it is critical that they can do all of this as quickly and as easily as possible. This requires designing the lobby to keep everything in a single Android activity.

Designing the lobby required balancing user experience with implementation difficulty. Originally, I had grand plans for the design of the lobby but when I started to implement it, I realized I could not implement my design and finish the rest of the project in time. I scrapped my original design and opted for something much simpler, throwing away any extra features and only implementing the essentials.

The new design features the song queue list prominently in the middle as it will be the most viewed and likely contain the most amount of information. Following convention, the music playback controls run across the bottom of the screen of the host device. Along the top, a list of currently connected users is displayed, with users being added or removed as they join or leave. There is also a button to bring up a screen for users to search and add songs to the queue.

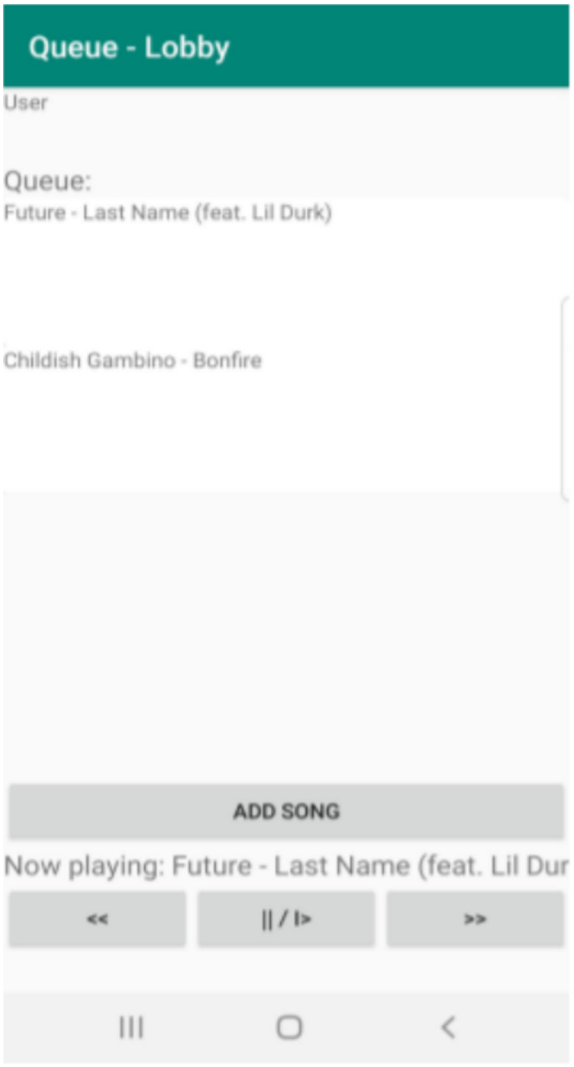

Figure 3: *The host's view of a Queue lobby, with a song playing and additional songs queued*

3.2 Implementation

Once a user has joined or created a lobby, the logic splits and differs depending on if the user is hosting or simply connecting to a lobby. As a host, the device is in charge of advertising the lobby, accepting and managing connections, accepting and responding to client input, and playing music. The host holds a reference to the NearbyHost class and uses this to advertise the lobby, manage connections to the lobby, and communicate with connected clients. Additionally, it holds a reference to a music controller, used to play/stop songs based on the chosen music provider for the lobby.

The client lobby is a much simpler implementation of the host lobby, holding only a NearbyClient connection used to communicate with the host. When a user performs actions that affect the lobby, such as adding a song or leaving the lobby, the client sends a message to the host. The host listens for messages from the client and takes the appropriate action based on the client request.

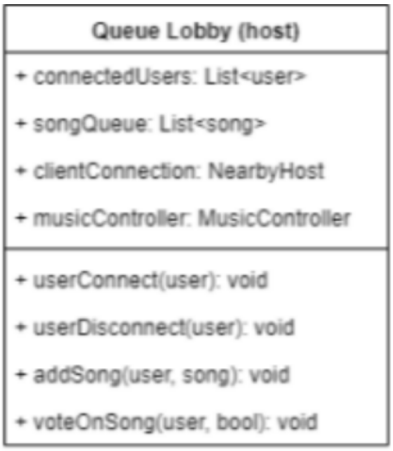

Figure 4: *A UML diagram of the Queue lobby implementation*

Since the lobby consisted of different segments of related widgets, I chose to implement these segments as Android fragments. Fragments allow you to design a section of your UI, and easily move it around and reuse it within your activities. The challenge, however, was designing the fragments to be able to communicate between each other and be reusable. To do this, fragments must not contain any references to their parent activity. Instead, fragments declare an interface with methods that the parent activity must implement where fragment functions are handled. This allows for reusability since the fragments are now totally independent, and for communication between fragments through their parent activity.

Chapter 4

Device communication

4.1 Design

Every user connected to a lobby should see the same information as everyone else. This requires communication between the devices to keep everyone in sync. I chose to accomplish this by having a host device that creates and is in charge of the lobby, while everyone else that connects is a client. All actions taken by the clients go through the host, who validates the action and notifies the other clients so that they are aware of the action and can update appropriately. The host needs to advertise itself and allow other devices to establish communication with it. Once communication is established, the client needs to send messages to the host to handle the client's actions, and the host needs to send messages to each client to keep them updated about other users' actions. A diagram depicting the structure of the connections between devices can be found below.

Figure 5: *A diagram depicting the connection between devices*

4.2 Implementation

Communication between devices is handled using Google's Nearby Connections API. The API allows for high-speed, encrypted peer-to-peer communication without having to worry about the details of establishing a connection. The API uses a combination of Bluetooth, BLE, and Wifi hotspots, leveraging the strengths of each while circumventing their respective weaknesses.

The API uses event listeners to handle actions taken when establishing and managing connections. To use the API, I had to implement handlers for these events. Each handler has computational logic that it needs to execute, as well as GUI logic that needs to execute to update the UI for the user. To keep my code organized and less coupled, I kept the GUI logic in the corresponding activity and the connection logic in a separate class.

To allow other users to join their party, the host needs to advertise that they have an open connection that a client can connect to. To find this open connection, clients have to actively search for nearby devices. If a client searches for open connections while the host is advertising, the client will be able to see the host in a list of nearby lobbies. At this point, the client can request a connection with a host from the list. The host will receive the connection and accept it, allowing the client to enter the lobby.

Chapter 5

Music Controller

5.1 Design

An important goal when designing this app was making it as accessible as possible. There already are many services for streaming music to your phone; I did not want to reinvent the wheel by trying to create my own streaming solution. Most of these streaming services have subscription fees and a music selection that may vary from another. Because of these factors, listeners usually have a preferred platform that they like to use. I did not want to turn away users because their preferred music streaming service was not available, so I decided to implement as many platforms as possible.

I did not want to worry about the details of implementing support for every music platform while working on the core features, so I decided to keep it generic. I chose one music streaming service and implemented features for that. After implementing all the features for that single platform, I created a generic version of it. The generic music controller is an interface containing methods to interact with a music platform, regardless of which one is used. This generic music controller can be used throughout the app to control the music without worrying about which platform the user is using. This not only keeps the codebase much more organized, separating the controller implementation from its usage, but also allows me to easily add more services in the future without changing existing code.

5.2 Implementation

The music controller is a class that handles interfacing with the chosen music service provider to let clients search for songs and the host to play them. It is also in charge of authenticating the user for the provider.

Currently, the only music controller implemented is for Spotify. The Spotify controller consists of two parts- the Spotify web API and Spotify Android SDK. The Spotify API is a RESTful web endpoint that can return JSON metadata about artists and songs in the Spotify Data Catalogue. In the Queue music controller, the API is used to search for songs so users can add them to the lobby queue.

Communicating with the API requires sending an HTTP request to Spotify over the web and parsing the JSON response. This is handled by Kaaes' Spotify API Android wrapper library that I used. The wrapper uses Retrofit as an HTTP client to communicate with the API, and creates Java interfaces from the response. This allows for quickly writing Android code that can communicate with the Spotify API.

The other part of the Spotify music controller is the Spotify Android SDK. The SDK is a library that allows your application to interact with the Spotify application running in the background as a service. It is a lightweight library that lets you control Spotify while all the actual work is done by the Spotify app itself.

In the Queue music controller, the Spotify Android SDK is used mainly for playback of the queue by the host. When a host presses the button to play a song, a message is sent to the Spotify app installed on their device that tells it to play the song. If the host has the Spotify app on their device, the song will start playing through Spotify.

The Spotify SDK also handles authentication. Before the host can play any song, they need to be logged in. When creating a lobby, the music controller will attempt to log in the user via the Spotify SDK. If the user is already logged in to the Spotify app on their device, they will be automatically logged in. Otherwise, they will receive a prompt to log in.

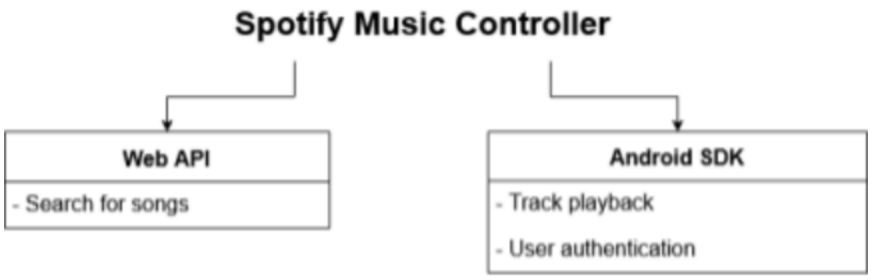

Figure 6: *A diagram depicting the two components of the Spotify music controller*

Chapter 6

Future Work

Currently, the application is in a very early phase. The core functionality is complete and users can create or join a lobby to collaborate on playlists together. Some features not yet implemented but planned for near-future work in order of priority are:

- Support for non-Android devices
- Improved UI
- Additional lobby features
- Additional streaming service platforms

6.1 Support for non-Android Devices

To keep it simple, I developed Queue on one platform alone. Since I was more familiar with Android and I only owned Android devices, I chose to develop for Android. My second priority for the future is to develop Queue for other devices as well, because what good is using the app if your friends can't use it too?

To port the app for iOS devices, it will have to be entirely rewritten in Swift. I am not familiar with Swift and would like to avoid having to keep two separate codebases for my application. The alternative is to rewrite the application using Xamarin, while will allow me to create a cross-platform application with a single codebase. Xamarin is also implemented in C#, which is another advantage because I am familiar with and enjoying development in that language.

I also want to implement support for desktop computers, most likely through a web application. This could be an alternative for iOS users instead of completely rewriting the app for iOS devices.

No matter how I choose to implement cross-platform support, it would involve changing how communication between devices is implemented. Currently, the host and clients communicate using a built-in Android API called Nearby Connections. This is

only available for Android devices so an alternative will need to be used for other platforms.

This item is at the top of my priority list because if I have to rewrite all or major portions of my application, it is better to do it as early as possible to keep it simple.

6.2 Improved UI

The UI is very basic with only the most necessary information being displayed and only core features implemented. Due to time constraints, the idea was to create a working project and then come back to revise and enhance the UI to be more visually appealing. Presentation is an important factor in an application's success so upgrading the UI to make it more visually appealing and fun to use is a big priority to me.

Currently, all the widgets in the application use the default Android theme, which makes the app look generic and boring. I would like to design a theme unique to my application, to make my app stand out more.

The layout of the screens was designed with ease of development in mind, not the user. Along with redesigning the theme of the widgets, I would like to also redesign the layouts. The new layout will be designed with the user in mind; the idea is to make using the application as intuitive and appealing as possible. Some additional research will

have to be conducted on standard Android layout conventions so that UI items are where the user expects, improving ease of use and user experience.

6.3 Additional lobby features

Currently, Queue lobbies only have core features implemented. Users can create/join a lobby, add songs, and the host can play them. To create a more engaging experience, lobbies need more features.

One big set of missing features is lobby administrative functionality- features that allow the host to do things like kick a user, manipulate the queue, and set lobby rules. This application is intended for use between friends so admin functionality wasn't a priority for the final prototype. However, when released for public use, it may be used in all types of settings. Administrative features for the host will give them control over their lobby and allow them to make it available to large groups of people without fear of anyone disrupting their lobby.

Some other features planned for implementation include letting users vote on a song to determine it's spot in the queue, better search function, and saving favorite songs.

6.4 Additional streaming platforms

Similarly to my choice to develop Queue solely for Android devices, I chose to develop the app with a choice of a single music streaming platform. I wanted to focus more on the features instead of worrying about implementation details that would differ from platform to platform. I developed the application with multiple platforms in mind though, creating and using an interface that can be extended for each new music platform. Adding new music streaming services is just a matter of implementing the interface.

References

[1] “Android SDK: Spotify for Developers.” *Android SDK | Spotify for Developers*, developer.spotify.com/documentation/android/.

[2] “Developer Guides : Android Developers.” *Android Developers*, developer.android.com/guide.

[3] Kaaes. “Kaaes/Spotify-Web-Api-Android.” *GitHub*, 19 Apr. 2016, github.com/kaaes/spotify-web-api-android.

[4] Overview | Nearby Connections API | Google Developers.” *Google*, Google, developers.google.com/nearby/connections/overview.

[5] “Web API.” *Spotify for Developers*, developer.spotify.com/documentation/web-api/.